REMARKS

Claims 1-21 are pending in the present application.

This Amendment is in response to the Office Action mailed January 19, 2001. In the Office Action, the Examiner objected to the specification, objected to Claims 11 and 18, and rejected Claims 1, 4-8 and 11-15, 18-21 under 35 U.S.C. § 102(e) and Claims 2-3, 9-10 and 16-17 under 35 U.S.C. § 103. Applicants have amended Claims 1, 8, 11, 15, 16 and 18. Reconsideration in light of the amendments and remarks made herein is respectfully requested.

I. SPECIFICATION

The Examiner objected to the Specification due to minor informalities. In response, Applicants have amended the Specification accordingly. Therefore, Applicants respectfully request the objection be withdrawn.

II CLAIM OBJECTIONS

Claims 11 and 18 were objected to because of informalities. In response, Applicants have amended Claims 11 and 18 accordingly to change claim dependency and to correct minor informalities. Therefore, Applicants respectfully request the objection be withdrawn.

III. REJECTIONS UNDER 35 U.S.C. § 102(b) AND § 103(a)

In the Office Action, the Examiner rejected Claims 1, 4-8, 11-15 and 18-21 under 35 U.S.C. § 102(e) as being anticipated by U.S. Patent No. 5,802,569 issued to Genduso et al. ("Genduso"). The Examiner rejected Claims 2-3, 9-10 and 16-17 U.S.C. § 103(a) as

A

being unpatentable over Genduso in view of U.S. Patent No. 5,309,451 issued to Noya et al. ("Noya"). Applicants respectfully traverse the rejections for the following reasons.

Gesundo discloses a computer system having cache pre-fetching amount based on CPU request types. The external cache (L2 cache) is a static random access memory (SRAM) (Gesundo, col. 3, lines 24-26).

Noya discloses a data and parity pre-fetching for redundant arrays of disk drives. The cache comprises 4 megabytes (MB) of random access memory (Noya, col. 4, lines 35-36). The cache is managed by a least recently used (LRU) algorithm (Noya, col. 4, lines 60-64).

Gesundo and Noya, taken alone or in any combination, do not disclose, suggest, or render obvious pre-fetching data to a cache queue. As the Examiner may be well aware, a queue is a first-in-first-out (FIFO) data structure. A queue is not a random access memory as disclosed in Gesundo and Noya. Furthermore, a FIFO process is distinctly different from an LRU algorithm. Since Noya discloses the LRU process to manage the cache, Noya teaches away from the present invention.

The above distinctions (i.e., standard SRAM and not queue for external cache, FIFO is different than LRU) are well known. The following are relevant excerpts from the books "The Cache Memory Book" written by Jim Handy ("Handy"), published by Academic Press, 1993 (pp. 14-22), and "Operating System Concepts", 5th edition, written by Abraham Silberschatz and Peter Baer Galvin ("Silberschatz"), published by Addison-Wesley, 1998 (pp. 304-308).

> "... This address is fed into the cache data RAM, which consists of a standard static RAM, and the data from that

addressed location becomes available to the CPU." (Handy, page 16).

"The key distinction between the FIFO and OPT algorithms... If we use the recent part as an approximation of the near future, then we will replace the page that has not been used for the longest period of time (Figure 9.11). This approach is the least recently used (LRU) algorithm." (Silberschatz, page 307).

In addition, Figure 1.8 in Handy shows that the external cache data memory is a standard RAM with address and data lines. In contrast, a queue has no address because the order of accessing is fixed.

Applicants are enclosing Appendix A which contains Handy and Silberschatz for reference.

Therefore, Applicants believe that independent Claims 1, 8, 15 and their respective dependent Claims are distinguishable over the cited prior art references. Accordingly, Applicants respectfully request the rejections under 35 U.S.C. § 102(b) and § 103(a) be withdrawn.

VERSION WITH MARKINGS TO SHOW CHANGES MADE

IN THE SPECIFICATION

The paragraph beginning on page 7, lines 8-14, have been amended as follows:

The peripheral devices 160_1 to 160_p are connected to the peripheral bus 140 to perform peripheral tasks. Examples of peripheral devices include a network interface and a media interface. The network interface connects to communication channel such as the Internet. The Internet provides access to on-line service providers, Web browsers, and other network channels. The media interface provides access to audio and video devices. The mass storage device [172] 150 include CD ROM, floppy diskettes, and hard drives.

IN THE CLAIMS:

The claims have been amended as follows:

1. (AMENDED) A method [to reduce latency in accessing a memory from a bus, the method] comprising:

pre-fetching a plurality of data from a memory to [the] a cache queue in response to a request; and

delivering the pre-fetched data from the cache queue to [the] a bus independently of the memory.

8. (AMENDED) An apparatus [to reduce latency in accessing a memory from a bus, the apparatus] comprising:

a pre-fetcher to pre-fetch a plurality of data from a memory to [the] a cache queue in response to a request; and

a cache controller coupled to the cache queue and the pre-fetcher to deliver the pre-fetched data from the cache queue to the bus independently of the memory.

11. (AMENDED) The apparatus of claim [8] 9 further comprising:

a peripheral bus controller coupled to the bus and the pre-fetcher to determine if the request is valid;

a data coherence controller coupled to the pre-fetcher to provide a purge signal when the request corresponds to a cache miss; and

a scheduler coupled to the request queue and the data coherence controller to store entries corresponding to the [requests] request, the entries being marked according to the purge signal from the data coherence controller.

15. (AMENDED) A system comprising:

a memory;

a bus; and

a bus access circuit coupled to the memory and the bus to reduce latency in accessing the memory from the bus, the bus access circuit including:

a pre-fetcher to pre-fetch a plurality of data from the memory to a cache queue in response to a request, and

a cache controller coupled to the cache queue and the pre-fetcher to deliver the pre-fetched data from the cache queue to the bus independently of the memory.

16. (AMENDED) The system of claim 15 wherein the pre-fetcher comprises:

a watermark monitor to determine if an amount of data in the cache queue is above a predetermined level;

a request packet generator coupled to the watermark monitor to place the request to a memory controller controlling the memory if the amount of data is not above the predetermined level, the request causing the memory controller to transfer the plurality of data to the cache queue; and

a request queue coupled to the request packet generator to store the request provided by the request packet generator.

18. (AMENDED) The system of claim [15] 16 wherein the bus access circuit further comprises:

a peripheral bus controller coupled to the bus and the pre-fetcher to determine if the request is valid;

a data coherence controller coupled to the pre-fetcher to provide a purge signal when the request corresponds to a cache miss; and

a scheduler coupled to the request queue and the data coherence controller to store entries corresponding to the [requests] request, the entries being marked according to the purge signal from the data coherence controller.

CONCLUSION

In view of the amendments and remarks made above, it is respectfully submitted that the pending claims are in condition for allowance, and such action is respectfully solicited.

Respectfully submitted,

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP

Dated: April 17, 2001

THINH V. NGUYEN
Reg. No. 42,034

12400 Wilshire Boulevard, Seventh Floor
Los Angeles, California 90025
(714) 557-3800

CERTIFICATE OF MAILING

I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail in an envelope addressed to: Assistant Commissioner for Patents, Washington, D.C. 20231 on: April 17, 2001

Rose Dunne — *Date* 4/17/01

APPENDIX A

THE CACHE MEMORY BOOK

JIM HANDY

ACADEMIC PRESS, INC.
Harcourt Brace & Company, Publishers
Boston San Diego New York
London Sydney Tokyo Toronto

This book is printed on acid-free paper. ♾

ACADEMIC PRESS, INC.
1250 Sixth Avenue, San Diego, CA 92101-4311

United Kingdom Edition published by
ACADEMIC PRESS LIMITED
24–28 Oval Road, London NW1 7DX

Library of Congress Cataloging-in-Publication Data

Handy, Jim.
The cache memory book / Jim Handy.
p. cm.
Includes index.
ISBN 0-12-322985-5
1. Cache memory. I. Title.
TK7895.M4H35 1993
004.5'3—dc20 93-6652
CIP

Printed in the United States of America
93 94 95 96 BB 9 8 7 6 5 4 3 2 1

instructions. This also means that the bus will be in use 40% of the time, the implications of which we will examine in Section 1.8.

The process of cacheing follows a certain pattern. At power-on, the cache contains random data and is disallowed from responding to a request from the CPU. When the processor reads data from a main memory location, the cache data RAM is ordered to copy that location's contents, while the corresponding cache directory location is told to copy the address requested by the CPU. This same sequence would occur for any further cycles until a loop was encountered. Once the processor reached the end of the loop, it would again output the address of the first location, which in all probability would still be in the cache data memory. This time, however, the data from that location would be supplied to the processor from the cache, rather than from the main memory, as would the rest of the instructions in the loop. It becomes pretty obvious how much of a help this would be. In our example program, the instructions in the loop would operate at main memory speeds only during the first go 'round, then would subsequently execute much faster from cache for the next 98 times through the loop.

This covers cached read cycles. What happens during write cycles depends greatly upon the cache policies chosen, and will be examined in depth in Chapter 2.

From the CPU's perspective, the data always comes from main memory, although at some times it arrives faster than at other times.

1.5 CACHE DATA AND CACHE-TAG MEMORIES

So we now know how simple it is to reroute the CPU's main memory request to be satisfied by the cache, but the design of the cache directory and cache data memories has yet to be explained. These might be pretty intimidating to the first-timer. The choice of directory architecture has a strong impact upon the design of the cache data memory, so the directory will be investigated first.

Certain college-level courses explain that the cache directory consists of a content addressable memory or **CAM**. This is a directory type which corresponds well to our example of the clerk in the credit bureau. A CAM is a backwards memory which outputs an address when data is presented to certain inputs. The address shows where a matching entry has been found within the CAM. All CAM locations are examined simultaneously for matching data, and if a match is found, its address is placed upon the address output pins. This is particularly useful when a 32-bit processor address needs to be converted to a much smaller cache address. The 32-bit processor address is presented to the data input pins of the CAM, and a

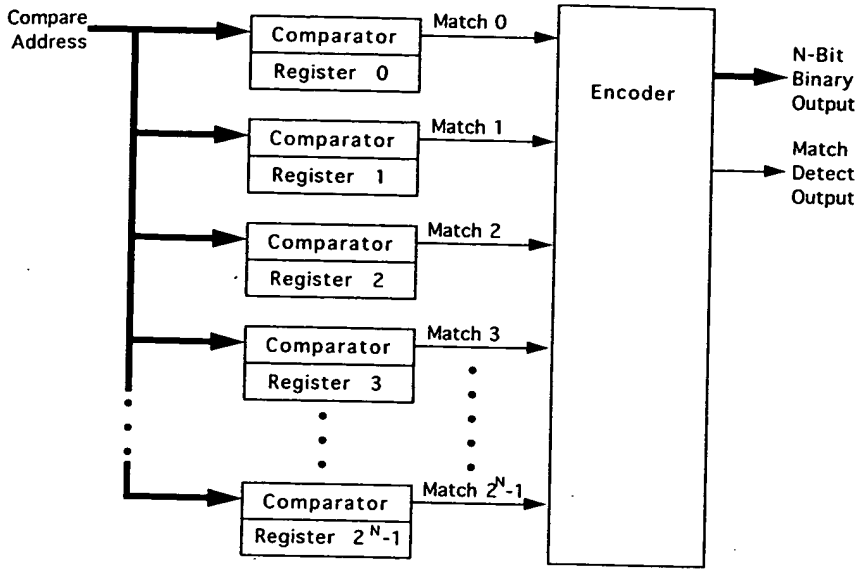


Figure 1.7. In a content addressable memory (CAM), an address presented to the compare address bus is compared simultaneously with the contents of every memory location. If there is a match at any location, the address of that location is sent to the N-bit binary output pins.

shorter address pops out of the address output pins. In truth, CAMs are almost never used to implement cache designs, but they facilitate the explanation of a cache directory, so we'll examine CAMs here first, then see what the alternatives are.

One big question that comes out of this is "what is a CAM?". A CAM is made of a series of address registers, each of which contains a comparator which compares the address contained within the register to the address currently on the compare address bus (see Figure 1.7). This is achieved by having a separate comparator attached to each and every address register. Although this sounds like a lot of hardware, it can actually be manufactured relatively easily in silicon by using a specialized, seven-transistor, static RAM cell which is less than twice as complex as the industry's standard four-transistor cell. The comparator outputs are encoded into a binary representation of the address of the matching entry. This somewhat esoteric part can be envisioned as a black box, much like any other memory. During a write cycle, both an address and data are presented to the CAM, and a write pulse is generated. During a read cycle, however, data is input to the CAM, and an address is output. The output address is the address of the CAM location which contains matching data. If no data match is found, a no-match signal is output from the CAM.

The reader might ask "Why aren't CAMs widely available?". The simple reason is that much simpler schemes perform nearly as well as the CAM approach, yet can be constructed from standard static RAMs and allow the use of less expensive cache data RAMs. These alternate approaches will be discussed here briefly and in depth in Chapter 2. Cache designers are not ready to pay as much for CAMs as CAMs would have to cost to attract the interest of semiconductor manufacturers. The author knows of only two kinds of CAMs available on the market today. The first is a mature product, an ECL 4 × 4 bit device which would have been designed into mainframe caches before faster (70ns) static RAMs became available. The other is a device designed specifically for LAN address filters, which requires a 48-bit compare address to be input in three multiplexed chunks over a relatively long period of time (i.e., 100ns).

Back to the example: Let's assume that the system has just been initialized and that no cache locations contain valid copies of main memory locations. Where in the cache should we put the first location accessed by the processor? Some processors seek their first instruction after a reset from the first main memory address, or address zero. Other processors start off at the top of the memory space (FFFF FFFF). Yet others seek a vector at one of these two addresses, then jump to the address pointed to by that vector.

If the cache directory is a CAM, then *any* main memory address can be mapped into *any* directory location. This approach is called **fully associative**, a term which simply means that any main memory address has the full freedom to be replicated anywhere within the cache. The method employed to determine which cache location should be used to store a copy of a main memory address is referred to as **mapping**, or **hashing**. (A fully associative approach is just one of many hashing algorithms which we will examine in this book.) An appropriate (hopefully unused) directory location would have to be assigned whenever something was to be put into the cache. The cache controller does this by assigning an address representing the directory location to be used to store the incoming data. This address is fed into both the CAM and the cache data memory while the main memory is being accessed. The data contained at the address of the main memory access (which is a completely independent value from the directory address just mentioned) is then written as data into the directory while the main memory data is both fed into the CPU and written into the cache data RAM (see Figure 1.8). Later, when the processor's address again matches the address stored in the CAM, the CAM outputs the address of the cache data RAM location which contains matching data. This address is fed into the cache data RAM, which consists of a standard static RAM, and the data from that addressed location becomes available to the CPU.

Partly due to the lack of availability of large CAMs, several ingenious alternatives to the CAM approach have been devised. These alternatives also

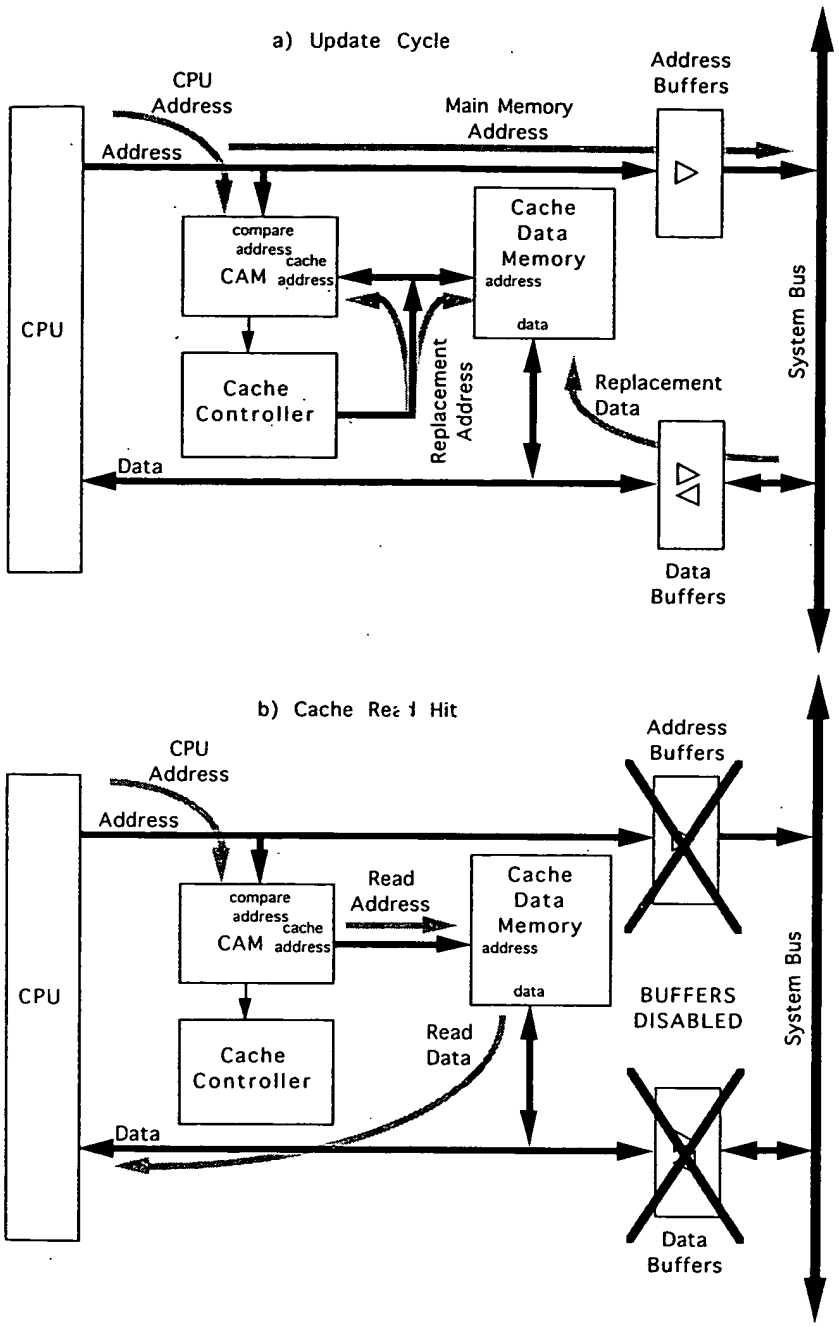


Figure 1.8. During the update of an entry in a fully associative cache (a), the cache controller feeds the address which is to be replaced into both the CAM and the cache data memory. When there is a cache hit (b), the CAM outputs this address to the cache data memory, causing its data to be fed into the CPU.

tend to be significantly less complex than the CAM approach. Returning to the file clerk analogy, let's put together an organizing system which will take advantage of the fact that the desk drawer is the same size as any of the drawers in any of the filing cabinets. The phone numbers used are all seven digits long and follow the format 867-5309. Furthermore, let's say that the filing cabinets are arranged so that each drawer represents a prefix (the first three digits of the telephone number), and each is capable of carrying the files for all 10,000 numbers or credit applicants within that prefix (from -0000 to -9999). So the first drawer would have numbers 000-0000 through 000-9999 and so forth. The clerk could also arrange the desk drawer into 10,000 slots, so that each slot would only be allowed to contain a file whose suffix (last four digits) matched the slot number; in other words, the number 867-5309 would only be allowed to be placed within desk drawer slot 5309. The directory would still be a paper on the wall containing 10,000 entries describing the phone numbers of the data stored in each of the desk drawer slots; however, each entry would be at a location which matched the suffix of the phone number, so only the prefix would need to be written into the directory (so for the number 867-5309, the prefix 867 would be written onto line 5309). Furthermore, the clerk would no longer have to choose where to put a new cache entry, so the directory entries would no longer need to contain the time of their last access. This is the most popular hashing algorithm in today's cache designs and is called **set associative**, since, for the sake of clarity, the last four digits of the telephone number have been named the cache's set address, and within any set the entry is associative (can have any prefix).

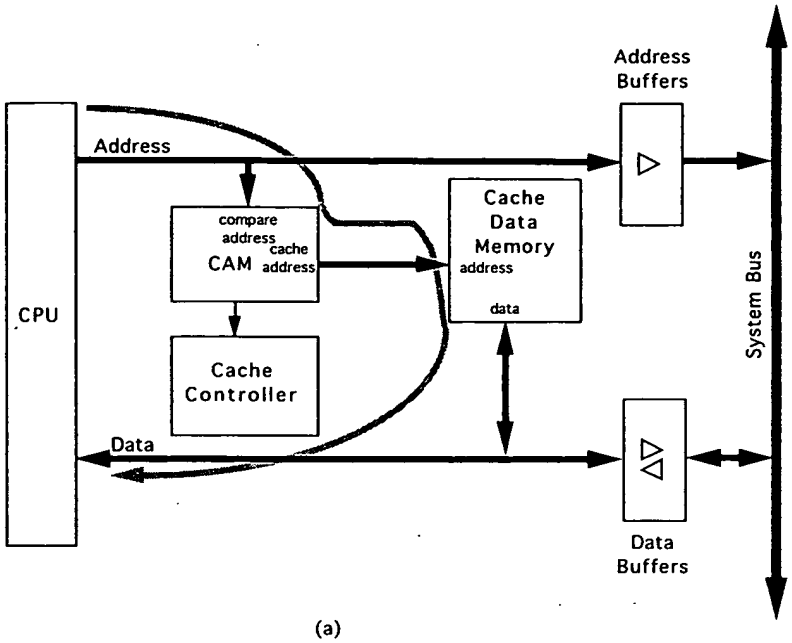
With this new method, the clerk can begin to look into the desk drawer while at the same time looking up the entry in the directory. There is no longer a need to cross-reference the directory to determine which desk drawer location contains the file. If the prefix (867) at the correct set address (5309) matches the one being requested by the caller, then the clerk will pull the file from desk drawer slot 5309 almost instantly. In a set-associative design, the address which is output from the processor is split into the equivalent of a prefix and a suffix at a location determined by the size and architecture of the cache. The lower address bits, those corresponding to the suffix or last four digits of the telephone number in the example, are called the **set** bits since they contain the set address. Some call these the **index** bits, but this terminology is in direct conflict with the terminology used in virtual memory translation, so, in the interests of clarity, it will not be used in this text. The remaining (upper) bits are compared against the directory entry for the selected set and are referred to as the **tag** bits.

You may recall a statement I made when first describing the clerk analogy, that this analogy does not account for spatial locality. This is easy to

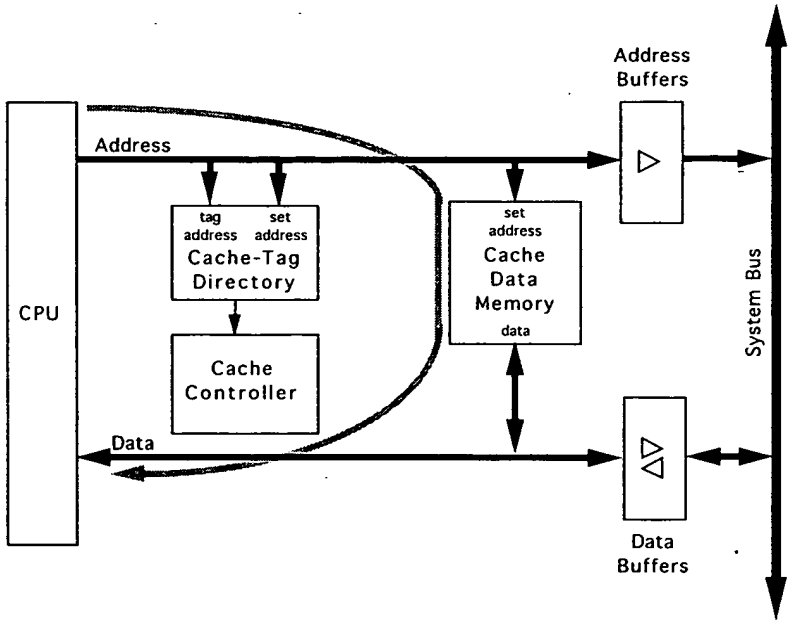
see, because the clerk might at any time receive calls for absolutely any file, representing any telephone number, in any order. Computer code never runs in such a random fashion, but tends to be full of loops and repetition, and whenever the program counter is not jumping around on a loop or a subroutine call, code will execute sequentially. The set-associative approach takes advantage of this spatial locality by placing sequential instructions not at entirely random cache locations, but at sequential locations. As just illustrated, the set-associative cache does not allow a main memory location to be mapped into any cache address, but instead the cache is constrained so that the lower address bits of the cache location must match the lower address bits of the matching main memory address. Since the processor will be running through a small group of sequential addresses, this looks as if it should pose no problem, and the set-associative cache should run every bit as efficiently as the more complex fully associative design. We will see otherwise shortly.

An earlier paragraph stated that a fully associative cache (one with a CAM) would require the use of more expensive cache data RAMs than a set-associative cache. Why? In a fully associative cache, all address bits are filtered through the directory before being fed to the cache data RAM. This means that during the cached CPU's fastest cycle, a cache read hit, the processor address must first travel through the directory CAM and be translated into a cache address, then that address must be passed through the cache data RAM before the read data can be made available to the CPU (see Figure 1.9a). This means that the delay between the processor's address output and the data returning from the cache is at least equal to the combined propagation delays of the CAM and the RAM. Thus, if data needs to be returned to the processor within, say, 50ns, then that 50ns must be divided between the CAM and the RAM. Today's 50 MHz microprocessors require this transaction to occur in 21ns, and today's fastest TTL I/O static RAMs are only as fast as 8ns. A system with an 8ns CAM (good luck!) and an 8ns cache data RAM would be hard-pressed to use any logic in this critical path.

The set-associative cache reduces this CAM plus RAM delay into a single RAM access. In Figure 1.9b, the CPU address is shown to run directly to both the cache directory and to the cache data RAM. For the sake of simplicity, the design shown uses a cache directory which is made of the same depth of standard static RAM as is the cache data RAM. (Alternatives to this will be explained in painful depth in Chapter 2.) Let's assume that the cache is 8,128 (8K) locations deep, requiring 13 address bits to be fully accessed. The lower 13 CPU address bits are then routed to both the cache directory and the cache data RAMs simultaneously. While the cache data RAM finds its replica of the data stored at a main memory location with



(a)



(b)

Figure 1.9. The critical speed path, from CPU address out to data in, runs through both the CAM and the cache data memory in fully associative cache designs (a). Set-associative caches (b) allow the CPU address to be fed directly to the cache data memory without going through any other devices.

matching lower address bits (but goodness knows which upper address bits), the directory looks at a replica of the upper address bits stored at the location with the same lower address bits. This is pretty confusing, but, based on the desk analogy, just remember that the last four digits of the telephone number 867-5309 are 5309, so slot 5309 is accessed in the file drawer. The directory then must tell us whether there is a match between the stored prefix (867) or whether some other prefix's number (984-5309) is being accessed. To settle the delay issue, though, notice that during the processor cycle time, the cache directory and the cache data RAM are being examined *simultaneously*, with the output of the cache directory being used simply to tell the CPU whether to proceed at full speed (a hit) or to wait while a main memory access is used to fetch the data which was requested by the CPU but did not reside within the cache.

The directory in a set-associative cache is considerably simpler than that of a fully associative cache. The fully associative cache's directory not only had to keep track of whether or not there was a valid entry for a main memory location contained within the cache (a fact which has not yet been described, but will be in Chapter 2), but also had to track where that entry resides in the cache data RAM, while the set-associative directory must only look to see whether the cache entry with the same set bits as the CPU's output address came from the location which has the same tag bits, and whether that location is valid. This is a simple comparison. Does or does not the tag at location 5309 match 867 for the phone number 867-5309, and is the location valid? Obviously, the tag comparison can be performed using exclusive ORs feeding into an AND gate, the function of a simple address comparator chip (Figure 1.10). Validity is equally as simple and will also be described in depth in Chapter 2. Set-associative caches assume (until told otherwise) during a processor read cycle that the data in the cache is what the processor is requesting. The cache data RAM starts the read cycle with its data outputs turned on, and its address pins will be tied directly to the processor's less significant address output pins (see Figure 1.9b). The cache-tag RAM, meanwhile, is looking up the address' more significant bits to see if the cache entry is from the correct address. If there is a match between the requested address and the address of the cached copy of the main memory data, then the cache controller gives the processor a Ready signal to accept the data from the cache data RAM and allows things to continue as they are. If the cache contains data at the same set address which is from a different part of main memory (i.e., the tag bits don't match), then the cache controller declares to the processor that a cache miss occurred by not asserting the Ready input, then reads main memory data into the cache, allowing the CPU to continue as the main memory data and address are being written into the CPU. For the 50ns processor cycle example just given, the cache data RAM in a set-associative cache need only have an access time of 50ns. The cache-tag

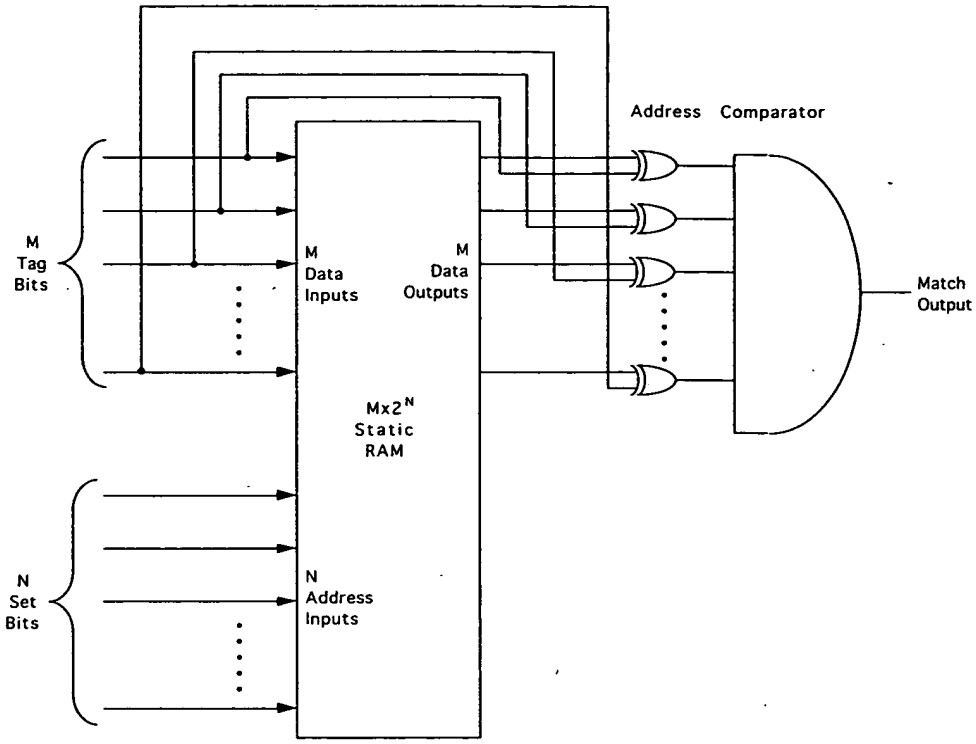


Figure 1.10. In a cache-tag RAM, the stored data at a certain RAM address (addressed by the set bits of the address bus) is compared against the upper address bits, or tag bits of the address bus. This comparison is performed with a simple address comparator consisting of exclusive OR gates feeding into an AND.

RAM would have a similarly slow access time, less the delays needed for the comparator and the downstream logic. Slower static RAMs are usually much less expensive than faster ones, so the designer can save a lot of money by going with a set-associative design. Some static RAM manufacturers provide products which have a comparator contained within a standard SRAM. This provides the same function at a certain advantage in speed, component count, and complexity of the downstream logic. Whether the directory is made from a simple RAM with a separate comparator or with a chip which includes both, the directory in a set-associative cache is called the **cache-tag** memory simply because it is where the address tags are stored. There are a few other terms for a cache-tag RAM, notably **tag RAM**, and **cache address comparator, address comparator, cache comparator**, or simply **comparator**.

Either of these is significantly less expensive than hard disk, but both have access times of several seconds. Since this level is removable, the size is limited only by the user's desire to keep an inventory of tapes or diskettes.

Cache memory fits between the first and second levels of the description. Several microprocessors today contain a small internal cache which is made of standard static RAM bits. Unlike registers, these locations cannot be written to and read from simultaneously, so certain operations will perform more slowly in on-chip cache than they would using internal registers. A register bit may require ten or more transistors to implement, whereas the SRAM cell used in the internal cache can be made of cells containing between four and six transistors per bit. Since the microprocessor is manufactured using the same chip, the on-chip cache is limited in size, mostly to keep the processor's cost from going through the roof. Die costs are not proportional to die size, but increase at more than a second-order rate. On-chip cache shares the advantage of not requiring an off-chip access at the expense of access time, so they are pretty fast. Today's microprocessors usually contain caches of between 256 and 8K bytes.

External caches are used both in systems which have and do not have on-chip caches. The external cache which is connected to an uncached processor serves to accelerate main memory accesses, as has just been explained. Such caches usually range from 32K bytes to 512K bytes and are made using economical SRAM chips constructed of four-transistor-per-bit memory cells. The cache's access time is around 10 to 25ns. If the processor has an on-chip cache, any external caches used will usually be implemented with architectures which differ considerably from the architecture of the on-chip cache, both to reduce costs and to make up for some of the deficiencies of the on-chip cache. These design issues will be examined in Chapter 2.

Table 1.1 shows in dollar magnitudes and bandwidth a typical memory hierarchy for a high-performance cached system, whose processor uses an on-chip cache.

Table 1.1. Memory hierarchy, and relative costs and typical sizes of each level.

Memory Type	Typical Size (bytes)	Cost per Megabyte ($)	Bandwidth (megabyte/sec)
Registers	10^1	10^5	500
On-chip cache	10^4	10^4	300
Off-chip cache	10^5	10^2	100
Main memory	10^7	10^1	50
Hard disk	10^8	10^0	10
Magnetic tape	Infinite	10^{-2}	0.2

Source: Dataquest 1992.

OPERATING SYSTEM CONCEPTS

Fifth Edition

Abraham Silberschatz

Bell Labs

Peter Baer Galvin

Corporate Technologies, Inc.

ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Editor-In-Chief: *Lynne Doran Cote*
Acquisitions Editor: *Maite Suarez-Rivas*
Associate Editor: *Deborah Lafferty*
Production Editor: *Patricia A. O. Unubun*
Design Editor: *Alwyn R. Velásquez*
Manufacturing Coordinator: *Judy Sullivan*
Cover Illustration: *Susan Cyr*

Access the latest information about Addison-Wesley titles from our World Wide Web site: http://www.awl.com/cseng

Reprinted with corrections, February 1998.

Library of Congress Cataloging-in-Publication Data

Silberschatz, Abraham.
Operating system concepts / Abraham Silberschatz, Peter Galvin. — 5th ed.
p. cm
Includes bibliographical references and index.
ISBN 0-201-59113-8
1. Operating systems (Computers) I. Galvin, Peter B. II. Title.
QA76.76.063S5583 1998

005.4, 3—dc21 97-28556
CIP

ISBN 0-201-59113-8
3 4 5 6 7 8 9 10 MA 01009998

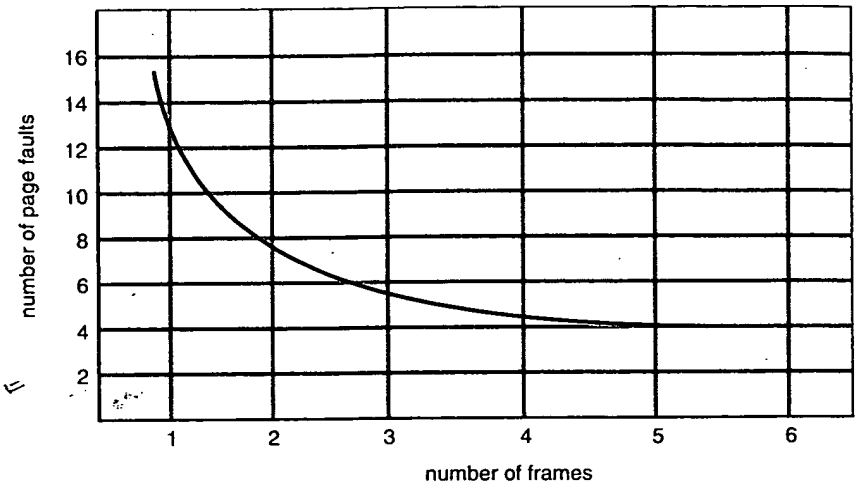


Figure 9.7 Graph of page faults versus the number of frames.

available. Obviously, as the number of frames available increases, the number of page faults will decrease. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults, one fault for the first reference to each page. On the other hand, with only one frame available, we would have a replacement with every reference, resulting in 11 faults. In general, we expect a curve such as that in Figure 9.7. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

To illustrate the page-replacement algorithms, we shall use the reference string

$$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

for a memory with three frames.

9.5.1 FIFO Algorithm

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults, and are brought into these empty

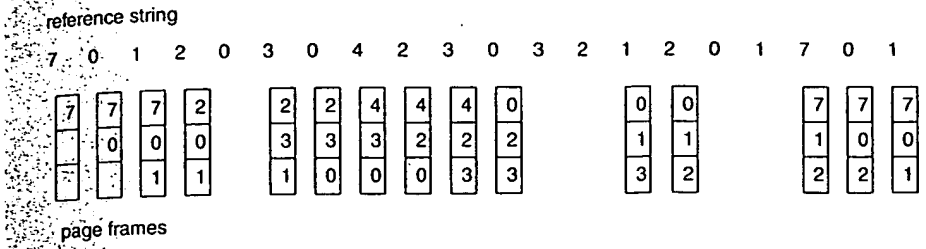


Figure 9.8 FIFO page-replacement algorithm.

frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1, and 2) to be brought in. This replacement means that the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 9.8. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. The page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution, but does not cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the reference string

$$1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.$$

Figure 9.9 shows the curve of page faults versus the number of available frames. We notice that the number of faults for four frames (10) is *greater* than the number of faults for three frames (nine)! This result is most unexpected and is known as *Belady's anomaly*. Belady's anomaly reflects the fact that, for some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

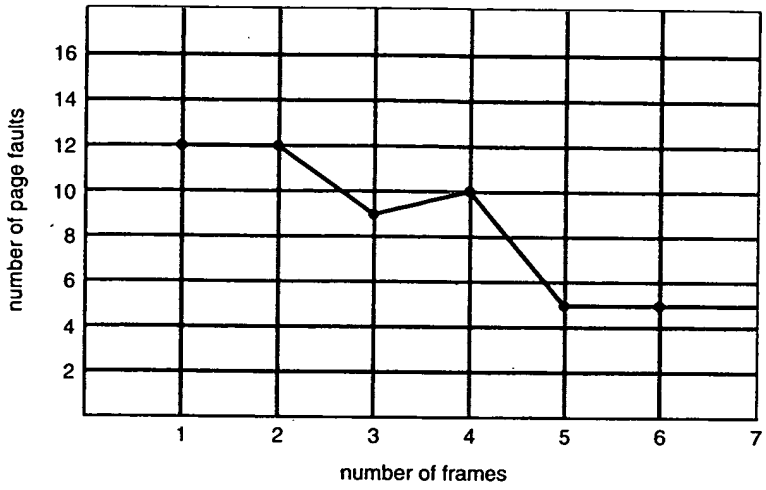


Figure 9.9 Page-fault curve for FIFO replacement on a reference string.

9.5.2 Optimal Algorithm

One result of the discovery of Belady's anomaly was the search for an *optimal* page-replacement algorithm. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal algorithm will never suffer from Belady's anomaly. An optimal page-replacement algorithm exists, and has been called OPT or MIN. It is simply

> Replace the page that will not be used
> for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 9.10. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with less than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (We

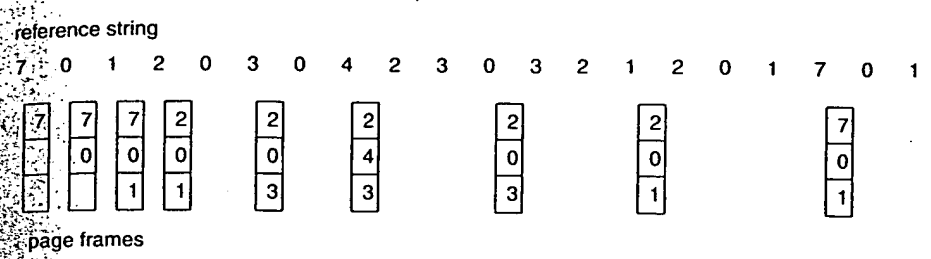


Figure 9.10 Optimal page-replacement algorithm.

encountered a similar situation with the SJF CPU-scheduling algorithm in Section 5.3.2.) As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be quite useful to know that, although a new algorithm is not optimal, it is within 12.3 percent of optimal at worst and within 4.7 percent on average.

9.5.3 LRU Algorithm

If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be *used*. If we use the recent past as an approximation of the near future, then we will replace the page that *has not been used* for the longest period of time (Figure 9.11). This approach is the *least recently used* (*LRU*) algorithm.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time. This strategy is the optimal page-replacement algorithm looking backward in time, rather than forward. (Strangely, if we let S^R be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on

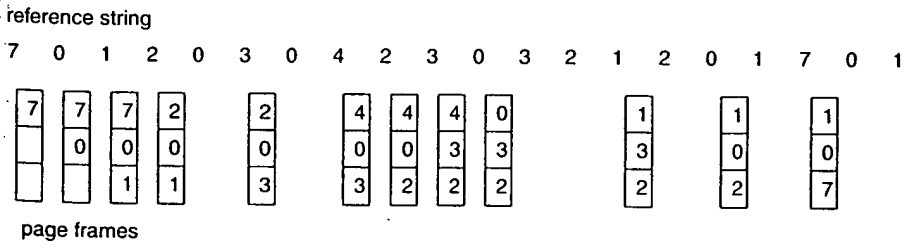


Figure 9.11 LRU page-replacement algorithm.

S^R. Similarly, the page-fault rate for the LRU algorithm on S is the same as the page-fault rate for the LRU algorithm on S^R.)

The result of applying LRU replacement to our example reference string is shown in Figure 9.11. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory {0, 3, 4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.

The LRU policy is often used as a page-replacement algorithm and is considered to be quite good. The major problem is *how* to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

- **Counters:** In the simplest case, we associate with each page-table entry a time-of-use field, and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page table for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page, and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

- **Stack:** Another approach to implementing LRU replacement is to keep a *stack* of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page (Figure 9.12). Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Neither optimal replacement nor LRU replacement suffers from Belady's anomaly. There is a class of page-replacement algorithms, called *stack algorithms*, that can never exhibit Belady's anomaly. A stack algorithm is an algo-